辅助文档

词法分析器

hash.h

`class hash_table<T>`

- `public`
  - `vector<T> table`

    存储哈希结构体，从0到n开始存储
  - `vector<int> h_table[HASH_MOD]`：

    哈希桶，存储哈希对象在 `table` 中的索引
  - `size`：

    元素数量
  - `hash_table()`：

    初始化
  - `clear()`：

    释放所有内存
  - `static int hash_str(char *s)`：

    给定字符串，返回哈希值
  - `find(const char s[])`：

    给定字符串，返回对应字符串在 `table` 中的索引
  - `insert(T)`：

    给定可哈希对象，返回对应哈希对象在 `table` 中的索引，可哈希对象必须实现 `get_hashobj()` 接口

`class hash_interface`

- `virtual char * get_hashobj()`：

  所有传入 `hash_table` 的对象都必须实现该函数，即返回一个字符串指针用于计算哈希值

buffer.h

- `BUF_SIZE`：

  内存缓冲区总大小
- `HALF_BUF_SIZE`：

  内存缓冲区的一半大小

`class buffer`:

- `protected`
  - `FILE * fd`:

    文件描述符
  - `char BUF[BUF_SIZE]`:

    内置内存缓冲区
- `public`
  - `void fill_half_buf(int)`: 用于填满内存缓冲区的一半，`part` 决定哪一半
  - `mover(int &r)`:

    用于安全向右移动指针，保证指针循环扫描且及时更新缓冲区
  - `bufcpy(char s[], int l, int r)`:

    复制内存缓冲区 $[l,r)$ 的区域到 `s` 中

sign.h

依托 `hash.h` 中的文件构造字符相关表格，定义了词法分析器的绝大多数工具

- 宏定义:

  定义了关于符号的宏名以及其符号编码
- `const int MAX_SIGN_SIZE=10`:

  最长符号名字大小
- `SIGN_NUM`:

  符号数量
- `MAX_IDF_NAME`:

  最长标识符长度
- `vector<PAIR> token`:

  `token[i].first` 为当前符号在 `code_table` 中的位置，
  `token[i].second` 为当前符号在 `sign_table` 中位置。对于系统默认字符串，如**关键字**、**运算符**、**分界符**
  第二项为 -1，只存储未出现过的字符串。
- `class sign`:
  声明 `sign`
- `class build_expr`:

  声明生成式
- `typedef hash_table<sign> sign_table`:

  声明符号表
- `typedef pair<int, sign> status_n_sign`:

  声明状态符号对
- `typedef my_stack<status_n_sign> stasign_stack`:

  声明状态栈

- `struct quad`:

  四元组

- `typedef vector\<quad\> quad_table`:

  声明四元组序列

- `typedef void(*action_fun)(quad_table *q_table, stasign_stack * sstack, build_expr * be, sign * sg, sign_table * s_table)`:

  声明函数指针类型，其为归约后的动作

- `typedef hash_table<category> code_table`:

  编码表，用于存储所有的类型与类型编码，
  实际中 `code_table.table[类型编码宏值]` 即为其对应的类型编码实例 `category`（需要配合 `init_code` 函数使用）

- `typedef hash_table<sign> sign_table`： 符号表
- `pair<LL, LL> PAIR`： 定义一个二元组
- `init_code(code_table &c_table)`： 通过预先定义好的类型宏名、类型宏值、类型名来初始化 `c_table`

`class category : public hash_interface`

类型编码类

- `public`
  - `char macro_name[MAX_SIGN_SIZE]`:

    宏名

  - `char category_name[MAX_SIGN_SIZE]`:

    实际名，若存在相关的代码实体，则该字符串与代码实体一致，比如 `int` 为关键字，则其实际名为 `"int"`，`"INT"` 为其宏名，宏值为 `INT=8`

  - `code`:

    宏的类型编码值

  - `get_hashobj()`:

    返回类型的实际名（`category_name`）

`class sign : public hash_interface`

用于存储词法分析器分析出的符号，包括关键字、标识符、字符常量、数值常量、运算符和分界符

- `sign_name`:

  对于所有未出现的单词（非系统内置字符串）都为其建立一个符号名，存储其在代码中的字符串。

- `code`:

  符号的类型编码，表征了符号类型

- `num`:

  存储数值常数

- `long long attr[10]`

  用于保存属性值

- `action_fun af`：

  归约函数指针，用来表示非终结符参与的归约式的语义规则

- `get_hashobj()`：

  返回 `sign_name` 作为哈希字符串

tool.h

- `LL s2d(char *s, int ind)`：

  根据传入进制将字符串转换为数字

- `double s2f(char*)`：

  将传入字符串转换为浮点类型

`class my_stack<T>`

相较于 STL 增强版的栈。

- `public`
  - `vector<T> table`：

    用于存储主体内容

  - `int top_pos`：

    栈顶指针

  - `my_stack()`：

    构造函数

  - `void pop()`：

    弹出栈顶元素

  - `void push(T)`：

    入栈一个元素

  - `T top()`：

    返回栈顶元素

  - `T top(int pos)`：

    返回从栈顶开始第 pos 个函数，pos 从 1 开始

  - `bool empty()`：

    返回栈是否为空

LA.

`class LA : buffer`

- `private`
  - `token tok`：

    词法分析出的 token 串

- `sign_table s_table`：

  词法分析出的符号表

- `code_table c_table`：

  词法分析依据的编码表

- `void read_digit(int &l, int &r, int attr)`：

  从缓冲区中读取指定进制数字，并将其放入 token 表中

- `void read_str(const char *s)`：

  从缓冲区中读取指定字符串并将其放入 token 表中

- `public`

  - `LA(FILE *fd, code_table c_table)`：

    传入读取的文件描述符以及依据的编码表

  - `void analysis()`：

    词法分析的主体部分，用于不断从指定文件中读取字符串并进行词法分析。

  - `void print_token(ILE* tf)`：

    打印出具有可读性的 token 表

  - `void print_sign_table(FILE *sf)`：

    打印出符号表

  - `void print_kw_table(FILE *kf)`：

    打印出关键字表，即所有系统内置字符串（包括符号）

H2 语法分析器

H3 build_expr.h

- `MAX_RIGHT_SIZE`：

  产生式右部符号的数量

- `typedef hash_table<build_expr> bexpr_table`：

  表达式表

H4 `class build_expr : hash_interface`： 存储产生式

- `public`

  - `char left[MAX_SIGN_SIZE]`：

    产生式左部字符串

  - `char right[MAX_RIGHT_SIZE][MAX_SIGN_SIZE]`：

    产生式右部字符串数组

  - `char hash[MAX_RIGHT_SIZE * MAX_SIGN_SIZE + MAX_SIGN_SIZE`：

    专门用于哈希的数组

  - `int size`：

    产生式右部的字符串数目

- `int ishashed`：

  是否已经哈希过了

- `char *get_hashobj()`：

  返回 `hash` 数组

- `void hash_init()`：

  根据产生式左右部初始化 `hash` 数组

- `static void parse(build_expr&, const char*)`：

  根据给定的字符串解析出对应的产生式

LR_table.h

- `const int END_SIGN_NUM=100`：

  终结符最大数目

- `const int STATUS_NUM=200`：

  最大状态数目

- `typedef hash_table<sign> msign_table`：

  非终结符表

- `const int MID_SIGN_NUM = 200`：

  非终结符最大数目

`class action`：

动作

- `public`
  - `char action`：

    动作，'r' 表示归约，'s' 表示移入，'a' 表示分析成功

  - `int num`：

    action为'r'时，表示第 num 个产生式，为's'时，表示加入的状态

`class action_table`：

动作表，存储所有动作

- `private`
  - `int status_num`：

    状态数目（行数）

  - `code_table * c_table`：

    其依据的编码表，用来识别终结符

  - `action actions[STATUS_NUM][END_SIGN_NUM]`：

    动作表，存储所有动作

  - `static inline int read_head(FILE*, char[])`：

    读取状态表第一行的终结符，如果到达行尾返回 0

  - `inline int read_body(FILE* fd, actions&)`：

读取状态表中一个单独的表格，如果遇到产生式，智能读取产生式并编号

- `public`
  - `bexpr_table * b_table`：

    产生式表，用来存储产生式
  - `void add_table(code_table*c_table, bexpr_table*b_table)`：

    连接编码表、表达式表
  - `void parse(FILE *fd)`：

    给定文件描述符，读入文件，并解析出动作表
  - `action get_action(int, int)`：

    给定状态与终结符编码，返回一个动作

`class goto_table`

- `private`
  - `int status_num`：

    状态数目
  - `msign_table *m_table`：

    非终结符表
  - `int gotos[STATUS_NUM][MID_SIGN_NUM]`：

    给定状态和非终结符编码，返回待压入的状态
- `public`
  - `void add_table(msign_table* m_table)`：

    连接非终结符号表
  - `static inline int read_head(FILE *, char[])`：

    读取 goto 表行首，即每一个非终结符
  - `static inline int read_body(FILE*, int&)`：

    读取 goto 表单元，即每一个整数
  - `void parse(FILE *fd)`：

    给定文件描述符，解析文件中的 goto 表
  - `int get_goto(int, char[])`：

    传入当前状态与非终结符号名，返回应入栈状态

LR.h

语法分析器的核心

应该终结符与非终结符使用同一个类型，符号中有标志其为终结符还是非终结符

- `typedef pair<int, sign> status_n_sign`：

  状态栈中具体的项，表示状态号与当前符号（终结符与非终结符都是 `sign` 类）
- `typedef stack<status_n_sign> stasign_stack`：

  状态符号栈

`class LR : buffer`

- `private`
  在 token 中根据索引找到 sign，与栈顶的状态进行配合，决定需要移入还归约，若移入则入栈，若归约则根据对应产生式进行归约（字符串匹配）
  生成非终结符，入栈，并根据goto决定入栈的状态编码，sign表符号与非终结符互不相干
  - `action_table * a_table`
    动作表
  - `goto_table * g_table`:
    goto 表
  - `sign_table *s_table`:
    终结符表
  - `msign_table *m_table`:
    非终结符号表
  - `token *t_table`:
    词法分析器产生的 token 串
  - `stasign_stack s_stack`:
    符号状态栈
  - `vector<int> reduce_seq`:
    生成式序列
  - `void shift(int, int)`:
    移入动作
  - `void reduce(int)`:
    归约动作
  - `static void wrong()`:
    发生错误
  - `static void accept()`:
    归约成功
- `public`
  - `quad_table q_table`:
    四元组序列
  - `void add(action_table*, goto_table*, sign_table*, msign_table*, token*)`:
    添加表的链接
  - `void analysis()`:
    对 `token` 串进行分析，并得到产生式序列
  - `void print_reduce_seq(FILE *fd)`:
    向指定文件中打印产生式序列

- `void print_quad_table(FILE *fd)`:

  向指定文件中打印四元组序列

中间代码生成器

action_fun.h

中间代码生成器的关键组成部分。其中为生成式编写了必要的动作。由于难以为特定生成式添加函数，因此实现的所有动作都基于非终结符。

- `static long long offset`:

  偏移量

- `int name_cnt`:

  变量名分配计数器

- `void gencode(quad_table *q_table, const char *op, const char *addr1, const char *addr2, const char *result)`:
  根据传入的运算符、操作数1、操作数2和结果存放地址，生成四元组并将其放入 `q_table`

- `char* newtemp()`:

  分配一个变量

- `wrong(const string& s)`:

  错误处理函数，当归约动作出现错误时转入此函数

- `void m_fun(*)`:

  对 M 进行归约

- `void type_fun(*)`:

  对 TYPE 进行归约

- `void ids_fun(*)`:

  对 IDS 进行归约

- `void n_fun(*)`:

  对 N 进行归约

- `void p_fun(*)`:

  对 P 进行归约

- `void assign_fun(*)`:

  对 ASSIGN 进行归约

- `void expr_fun(*)`:

  对 EXPR 进行归约

- `void tt_fun(*)`:

  对 TT 进行归约

- `void ff_fun(*)`:

  对 FF 进行归约

汇编代码生成器

CG.h

- `const int MAX_NAME_SIZE = 100`

  最大变量名长度

- `REG_NUM = 4`

  寄存器个数

- `const char reg_name[4][10]`

  寄存器名

`class variable : hash_interface`

用于存储中间代码中的变量及其相关信息

- `public`
  - `char name[MAX_NAME_SIZE]`

    变量名

  - `int r`

    `r` = -1 表示该变量不存在于任何一个寄存器，否则表示该变量在第 `r` 个寄存器

  - `int refcnt`

    表示该变量被其后指令引用的次数

  - `vector<int> pos`

    存储所有引用该变量的指令编号

  - `char *get_hashobj()`

    返回哈希值

- `typedef hash_table<variable> variable_table`

  变量表

- `struct instruction`

  用于存储汇编指令，包括操作符、参数1、参数2

`class CG`

汇编代码生成器

- `private`
  - `int reg[REG_NUM]`

    `reg[i]` 存储其存储的变量在变量表中的索引

  - `variable_table v_table`

    变量表，存储每个变量名称的相关翻译信息

  - `quad_table *q_table`

    指令缓存

  - `vector<instruction> inst_seq`

生成的汇编指令序列

- `void insert_v_table(int mccnt, char s[])`

  传入变量在四元组序列中的引用位置以及变量名称，将相关信息加入到 `v_table` 中

- `int get_reg(char *p1, char *res)`

  四元组形式为（op, p1, p2, res），传入 `p1` 和 `res` 得到空闲寄存器资源或抢占寄存器资源

- `void updatev(char *p1, char *p2, char *res)`

  当处理完代码之后，更新每个变量在变量表中的信息

- `void move_inst(char *p1, char *res)`

  生成 `move` 指令

- `void bino_inst(char *op, char *p1, char *p2, char *res)`

  生成二元操作数指令

- `static void wrong(char *s)`

  错误处理函数

- `public`
  - `void add(quad_table* qt)`

    添加由中间代码生成器产生的中间代码序列

  - `init()`

    根据中间代码序列将相应变量添加至变量表中，并初始化所有变量信息

  - `analysis()`

    得到汇编指令序列

  - `print_inst_seq(FILE *fd)`

    向指定文件中打印汇编指令序列